SUPPORTING FAULT-TOLERANT COMMUNICATION IN NETWORKS

A Thesis

by

KHUSHBOO KANJANI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2008

Major Subject: Computer Engineering

SUPPORTING FAULT-TOLERANT COMMUNICATION IN NETWORKS

A Thesis

by

KHUSHBOO KANJANI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Co-Chairs of Committee,	Jennifer Welch
	Alexander Sprintson
Committee Member,	Anxiao Jiang
Head of Department,	Valerie E. Taylor

December 2008

Major Subject: Computer Engineering

ABSTRACT

Supporting Fault-Tolerant

Communication in Networks. (December 2008)

Khushboo Kanjani, B.Tech., Indian Institute of Technology Roorkee, India

Co–Chairs of Advisory Committee: Dr. Jennifer Welch
Dr. Alex Sprintson

We address two problems dealing with fault-tolerant communication in networks. The first one is designing a distributed storage protocol tolerant to Byzantine failure of servers. The protocol implements a multi-writer multi-reader register which satisfies a weaker consistency condition called MWReg. Most of the earlier work gives multi-writer implementations by simulating m copies of a single-writer protocol where m is the number of writers. Our solution gives a direct multi-writer implementation and thus has bounded message and time complexity independent of the number of writers. We have simulated the complete protocol to test its performance and also proved its correctness theoretically.

The second problem we address is of providing a reliable communication link between two nodes in a network. We present a capacity reservation algorithm in the case for upper bounds on edge capacities and costs associated with using per unit capacity on any edge. We give a flow based approximation algorithm with cost at most four times optimal.

To conclude, we design a distributed storage protocol and a capacity reservation algorithm which are tolerant to network failures.

To My Family

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my advisor, Dr. Jennifer Welch, for introducing me to research in the area of distributed computing, for being so patient and supportive, especially in the not so good times, and for the detailed feedback on my reports which helped me improve my writing skills. This thesis was not possible without your guidance and support.

I would also like to thank the members of my advisory committee, Dr. Alex Sprintson and Dr. Anxiao Jiang, for their guidance in research and the intriguing questions about my work. Thanks to the staff in the Department of Computer Science for making my academic life here so convenient, to AWICS members for being so good mentors and friends.

A special thanks to Sonal Tyagi, Vidhi Jain and Soumya Kulkarni for being there always, to Ganesh Krishnan, Fasihullah Askiri, Ashmesh Mehra, Ullas Sankhla, Swaroop, Sandeep Yadav and Parimal Parag for their invaluable advice at various steps of my graduate career, and to Champa Joshi for being a wonderful roommate for two years of my stay here in College Station.

Finally, I am thankful to my parents, my younger sister and cute little brother for their love and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A. Overview

With the ever increasing applications of computer networks, there is a need to design better algorithms for communication. Fault-tolerance is one important design aspect as the network components become more prone to failures with the increase in size and usage. A network can be modeled as a graph with nodes and edges. Any node or edge in the network may fail. These failures are broadly divided into two categories : (a) crash (b) Byzantine. In crash failures the component becomes unresponsive, but in Byzantine failures the component can return corrupted data.

We deal with both kind of failures in two independent problems. The first one is designing a distributed storage system tolerant to Byzantine failure of some nodes. A distributed storage system stores data at multiple nodes and can be accessed concurrently by multiple clients. The nodes and clients can be geographically at different locations. With the increase in internet bandwidth, clients can communicate with the nodes in real time. This makes a distributed storage system most suitable for large scale data storage, retrieval and search.

We implement the distributed storage system by a set of servers. One way to formulate this system is as an implementation of a multi-writer multi-reader register. The semantics of such a register are defined by a consistency condition. Two of the common consistency conditions are atomicity and regularity. There is a tradeoff between the strength of a condition and the cost of implementing it. We discuss this in more detail in Chapter II. Atomicity is a strong condition and so multi-writer

---

The journal model is *IEEE Transactions on Automatic Control.*

atomic protocols are often complicated and expensive. One way to alleviate this problem is to look at weaker consistency conditions like regularity which are still potentially useful. There has been previous work on multi-writer regularity but it was not fault-tolerant. The objective of our research is to develop a fault-tolerant implementation of one multi-writer regularity definition. Intuitively because a weaker condition is being implemented, the new algorithm is cheaper than the known multi-writer atomic algorithms.

The second problem we address is of reserving capacities on edges in a network to provide a resilient communication path between two nodes in the network. There is copious literature in the area of finding min-cost flows, capacity reservation, etc., but very few results are tolerant to failures. In this work, we deal with crash failure of edges. The challenge is to minimize the total cost of capacities reserved as each edge has a per unit cost of usage. This problem has lately gained attention in the design of virtual private networks [1].

B. Contributions

The contribution of our research is to give a fault-tolerant implementation of a multi-writer multi-reader regular register. To the best of our knowledge, this is the first direct implementation of a multi-writer register which satisfies one of the definitions of regularity for multiple writers and is tolerant to Byzantine failure of servers. Our solution has bounded message and time complexity and can tolerate Byzantine readers. There is no upper bound on the number of writers and the storage cost at the servers is constant. Simulation and theoretical analysis proved the correctness and desired behavior of the algorithm.

We also present an approximation algorithm to reserve capacities on edges in a

network. The aim is to ensure a reliable communication link of two units of capacity between a source and a destination. At most one edge in the network can fail at any time. Also each edge has an upper bound on the amount of capacity that can be reserved and the capacities are to be reserved in integral amounts. Our solution had a total cost of at most 4 times the optimal solution. To the best of our knowledge, this is the best know algorithm for this problem.

CHAPTER II

DISTRIBUTED STORAGE

Distributed storage systems have become a de-facto standard for handling the enormous amount of data on the internet. Some of the issues in these systems are availability, high throughput and fault-tolerance. Highly available systems aim to maximize the time the system is accessible during a given measurement period while throughput is a measure of the amount of requests handled per unit of time. We focus on the fault-tolerant aspect in this work. A fault-tolerant storage system guarantees data availability and integrity in the presence of failures. The two main approaches for designing fault-tolerant systems are replication and erasure coding which are described below.

Replication: In a scheme using replication, complete data is stored at multiple servers. When the register value is changed, it has to be updated at all servers which adds to the communication cost. One of the approaches to reduce this communication cost is to use Quorum systems. Quorums are subsets of servers such that the intersection of any two subsets is non-empty. Each read or write chooses a quorum and accesses only the servers in that quorum.

Erasure Coding: Erasure codes split data into blocks such that a fraction of those blocks can be used to reconstruct the original data. In a storage scheme based on erasure coding, each server stores exactly one block of data. Since servers do not store complete data, this scheme has less storage cost as compared to a replication based scheme. But it is more difficult to implement especially in the case of Byzantine failures.

A distributed storage system can be formulated as an implementation of a shared read/write register over an underlying network of server nodes. A shared register can be accessed by multiple processes at the same time.

The behavior of such a shared register is defined by a consistency condition which is a set of constraints on values returned by data accesses when those accesses may be interleaved or overlapping. A strong consistency condition like atomicity (or linearizability) [2] gives an impression of sequential behavior and so it has a high implementation cost in terms of message and time complexity. Weaker consistency conditions have lesser implementation cost but can be difficult to program with. For the case of single writer, Lamport [2] defined three consistency conditions in increasing order of strengths which are safe, regular and atomic. A **safe** register returns the value of the latest preceding write in case of no ongoing writes. There is no guarantee on what value the register will return when a write is concurrent with the read. A **regular** register returns either an ongoing write's value or the last completed write's value. An **atomic** register gives an impression of sequential behavior. The consistency conditions safe and regular cannot trivially be extended to the multi-writer case because the definition of "latest preceding write" is not clear. Shao et al. in [3] formally extend the definition of regularity in many possible ways for multiple writers. Our objective is to implement a multi-writer register satisfying one of those weaker consistency conditions and tolerant to failures. The underlying model is a set of servers which communicate by message passing. A fraction of these servers could become non-responsive (i.e. crash failure) or arbitrarily corrupted (i.e. Byzantine failure). Our approach for fault-tolerance is replication based.

A multi-writer register is more challenging to implement than a single-writer because of the following issues

- Deciding timestamp of a new value to be written is not trivial unlike the single writer case.
- With more than one writer updating the server's data concurrently and also a fraction of servers being Byzantine, the reader's protocol gets complicated.

Because of these reasons, the majority of the fault-tolerant implementations of multi-writer registers are simulations of a single-writer protocol. In such a simulation each writer's written value is stored as a separate variable at the server side. Correspondingly a reader reads values from all these variables and chooses the one with the latest timestamp. So the communication cost for the read is $O(m \times R)$ where m is the number of writers and R is the reader's communication cost in the single-writer protocol. In addition to that, the storage cost at the servers is $O(m)$ as compared to the single writer case where it is $O(1)$. It is apparent that there are two major limitations of this scheme:

- Implementation cost is proportional to the number of writers.
- There is an upper bound on the number of writers allowed.

To overcome these limitations, we focus on the direct implementation of a multi-writer register in this thesis.

The next section discusses the preliminaries and definitions of terms used in subsequent sections.

A. Background

A multi-writer multi-reader register supports two operations, *read* and *write* which can be executed by any client process. The response to a read operation is the returned value and to a write operation is the acknowledgment (ack).

The behavior of a shared register in the presence of multiple accesses is defined with respect to the desired behavior of the sequential register. The sequential specification of a read/write register is the set of all sequences of read and write operations such that each read operation returns the value of the latest preceding write operation; if there is no preceding write, then the read returns the initial value of the register.

Following are the definitions of terms from [3] which are used in the subsequent sections.

Definition 1. *A sequence of operations on a shared object is* **legal** *if it belongs to the sequential specification of the shared object.*

If σ is a sequence of operation invocations and responses, $\sigma|i$ denotes the subsequence of σ including only all the invocations and responses performed by process p_i.

Definition 2. *A sequence σ of invocations and responses is a schedule if, for each i, $0 \leq i < n$, the following hold:*

- *$\sigma|i$ consists of alternating invocations and matching responses, beginning with an invocation; and*
- *if the number of steps taken by p_i is finite, then the last step by p_i is a response, i.e., every invocation has a matching response.*

Finally, let $writes(\sigma)$ denote the set of all write operations in schedule σ. A partial order $<_\sigma$ on operations in σ is defined as follows: For two operations o_1 and o_2 in σ, $o_1 <_\sigma o_2$ if and only if the response of o_1 precedes the invocation of o_2 in σ.

Definition 3. σ-consistent: *Given a schedule σ, a permutation π of a subset of*

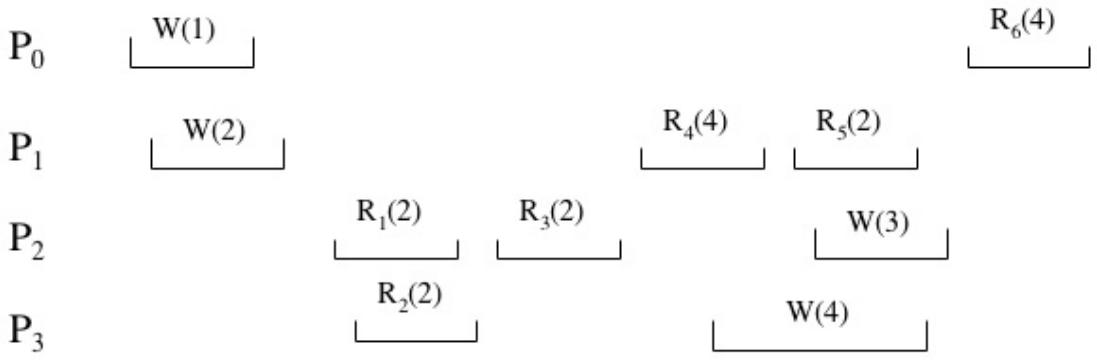

Fig. 1. A schedule that satisfies MWReg

ops(σ) *is* σ*-consistent if, for any operations* o_1 *and* o_2 *in* π, o_1 *precedes* o_2 *in* π *whenever* $o_1 <_\sigma o_2$.

Some more terms need to be explained in order to define MWReg. A write is **relevant** to a read if the invocation of the write is before the response of the read. For example, in the schedule in Fig. 1 the writes "relevant" to the read R_4 are W(x,1), W(x,2) and W(x,4). Informally MWReg requires that any pair of read operations agree only on the ordering of write operations that are "relevant" to both of them. $writes_{\leftarrow r}(\sigma) = \{w | w \in writes(\sigma)$ and w begins before r ends in σ $\}$.

Definition 4. MWReg: *A schedule* σ *satisfies MWReg if there exists a permutation* π *of ops(*σ*) such that, for all read operations r in ops(*σ*), the projection* π_r *of* π *onto* $writes_{\leftarrow r}(\sigma) \bigcup \{r\}$ *satisfies :*

- π_r *is legal, and*
- π_r *is* σ*-consistent.*

A shared memory object satisfies MWReg if all schedules on that object satisfy MWReg.

Fig. 1 gives an example of a schedule that satisfies MWReg. The permutations of "relevant" writes for all read operations is as follows:

R_1: W(1), W(2), $R_1(2)$

R_2: W(1), W(2), $R_2(2)$

R_3: W(1), W(2), $R_3(2)$

R_4: W(1), W(2),W(4), $R_4(4)$

R_5: W(1), W(2), $R_5(2)$, W(3), W(4)

R_6: W(1), W(2), W(3), W(4), $R_6(4)$

MWReg conforms with the singler-writer definition of regularity because the two conditions on the projection π_r for all read operations ensure that any read operation returns a value written by either an overlapping write or a precdeing write. MWReg can be seen as a multi-version of regularity because it makes sure that any pair of read operations have a common view of the write operations which are concurrent or preceding both of them. The read operations need not agree on the ordering of the write operations which happened later and so this condition is weaker than atomicity. The example schedule in Fig. 1 does not satisfy atomicity because there is no permutation of the operations $R_4(4)$, $R_5(2)$, W(3) and W(4) which is legal and σ-consistent.

B. Related Work

Distributed storage systems have been an important subject of research. Some papers prove impossibility or lower bound results. Others give protocols that can be categorized based on the communication model, assumptions and desired behavior. We look mainly at fault tolerant protocols. In the replication based approach, there

are two directions of study. The first one is complete replication in which the results aim at achieving Byzantine tolerance optimally (i.e. for $n \leq 3f + 1$), while in the quorum based approach more focus is given to high availability.

The lower bound results proved in the area of fault-tolerant shared registers are discussed here:

Number of Servers: Any fault-tolerant storage protocol requires at least $3f + 1$ servers to ensure safe semantics and liveness. This lower bound was proved in [4] and it holds true for randomized protocols and self-verifying data (data that cannot be undetectably altered, e.g. digitally signed data) also.

Time Complexity: In the asynchronous model, we measure time complexity in terms of rounds. One round is defined as the maximum time delay in one round-trip of communication between any two nodes. A lower bound of 2 rounds has been proved for both the read and write protocols for $n \leq 4f$.

- It is impossible to emulate the READ operation of a safe Single-Writer Single-Reader (SWSR) wait-free storage by invoking a single round of operation on base objects when $n \leq 4f$ [5].
- It is impossible to emulate the WRITE operation of a safe SWSR wait-free storage by invoking a single round of operation on base objects when $n \leq 4f$ [6].

In Table I, we compare the relevant papers based on the following parameters:

- **Wait-Free:** Implies that the client protocol is wait-free i.e. any client can complete any of its operations regardless of the failures of other clients.
- **StoS:** "Yes" in the table column implies that no server to server communication is required by the protocol.

- **Atomic:** Indicates the protocols which satisfy atomicity.
- **3f+1:** It has been proved that $3f+1$ is the minimum number of servers required to design a fault-tolerant protocol where f could be Byzantine. So a "Yes" in the corresponding column indicates that the solution is optimally resilient.
- **Multi-Writer:** This indicates whether the solution can handle concurrent writes.
- **BRounds:** This stands for Bounded Rounds. Some of the earlier results had unbounded message and time complexity in the worst case.

Most of these papers like [7, 8] give multi-writer register implementations by simulating m copies of the single-writer protocol where m is the number of writers. We focus on designing a multi-writer register directly without using multiple copies of a single writer register. Another aspect is that the majority of the work focuses on implementing atomic registers. Atomicity is the strongest consistency condition and so has higher implementation cost compared to weaker conditions like regularity. In this work we focus on designing a register which satisfies a multi-writer version of regularity called MWReg.

C. Model

The model we consider is a network of servers and clients connected by an asynchronous message passing layer as shown in Fig. 2. An asynchronous network does not expect any synchrony between the network components and puts no upper bound on message delays. There are n servers and unbounded number of clients. Servers store the data and the clients communicate with servers to read and write the data. Such a model is suitable for wide-area networks, since there are no timing assump-

Table I. Related work in fault-tolerant distributed storage

$Paper$	$Wait-Free$	$StoS$	$Atomic$	$3f+1$	$Multi-Writer$	$BRounds$
MAD02[4]	-	-	Yes	Yes	-	-
BD04[8]	-	-	Yes	-	Yes	-
ACKM04[6]	Yes	-	-	Yes	-	-
BD06[9]	Yes	-	Yes	-	-	-
CT05[10]	Yes	-	Yes	Yes	Yes	Yes
AAB07[7]	Yes	Yes	Yes	Yes	Yes	Yes
RQS07[11]	Yes	Yes	Yes	Yes	-	-

tions on the delay in passing a message. We assume reliable, FIFO channels between clients and servers and at most f servers can be Byzantine where $n > 3f$.

D. Algorithm

We modify the algorithm in [7] to give an efficient multi-writer implementation for a weaker consistency condition called MWReg (Definition 4). The algorithm is wait-free, tolerant to Byzantine servers and can handle unbounded number of clients. It cannot tolerate Byzantine writers. The earlier solutions for multiple writers had storage and communication costs proportional to the number of writers. Our solution has a constant storage cost at the servers.

The value accessed from the register is represented as a tuple $\langle v, ts, wid \rangle$ where v is the value, ts is the timestamp value and wid is the writer id. Since the register is accessed by multiple writers, we use the pair $\langle ts, wid \rangle$ to get a unique ordering of the timestamps. A timestamp $TS_1 = \langle ts_1, wid_1 \rangle$ is greater than $TS_2 = \langle ts_2, wid_2 \rangle$ if

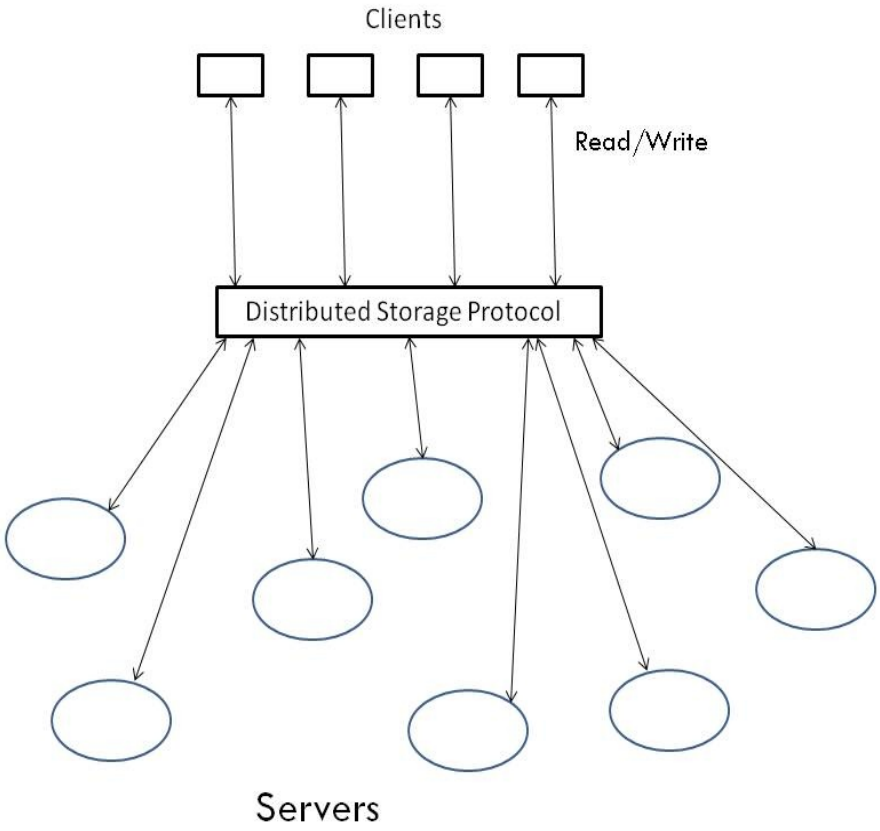

Fig. 2. Communication model

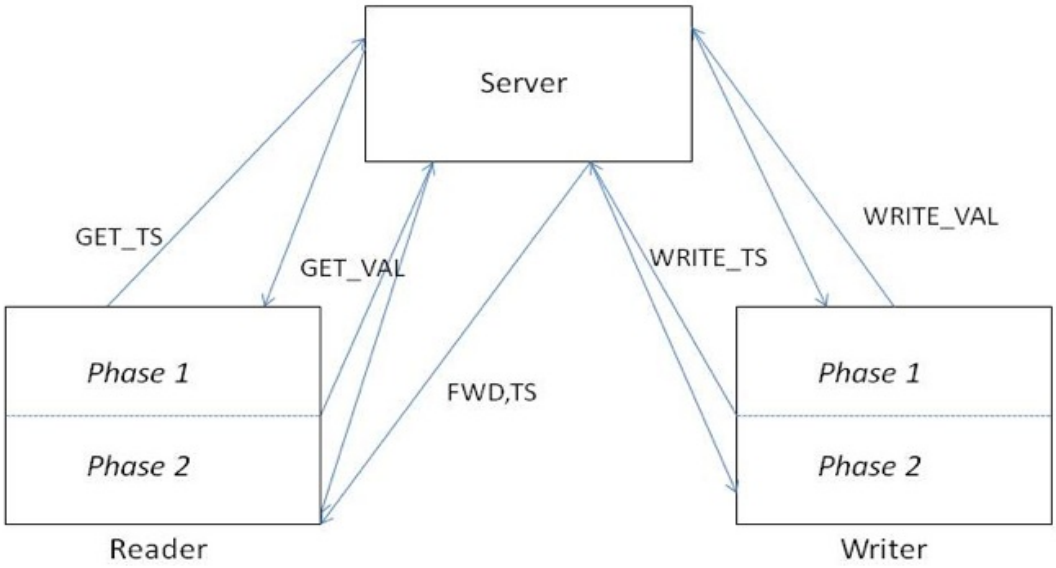

Fig. 3. Overview of the algorithm

and only if:

$$ts_1 > ts_2 \text{ OR } (ts_1 == ts_2 \text{ AND } wid_1 > wid_2) \tag{2.1}$$

An overview of the communication between the servers and clients in shown in Fig. 3. The figure demonstrates the messages exchanges that take place between the reader/writer and the servers. In what follows, we give a detailed description of the reader, writer and server protocol.

1. Reader

A client reader maintains three arrays **timeStamps**, **Fwd** and **Values** during the execution of the read protocol. It resets these arrays when a read returns. The read protocol described in Algorithm 1 is two phase. In the first phase, the reader requests timestamp by sending a GET_TS message to all servers. When it receives $n - f$ responses, the reader goes to Phase 2. If in Phase 1, a reader receives a FWD

message, it accepts it and updates its **Values** and **Fwd** array.

In Phase 2, the reader requests values from servers by sending a GET_VAL message. It continues to update its **Values** and **Fwd** arrays when it receives a FWD or VAL message. The reader continues to accept all messages till the termination condition (mentioned in Algorithm 2) is satisfied. Termination is possible in two ways:

notOld() and valid() The notOld() condition makes sure that an old timestamp is not chosen. This is done by choosing a timestamp which is greater than or equal to at least $2f$ received timestamps. Why $2f$? Because any writer waits for acknowledgment from $n - f$ servers. So f good servers might not be updated with the last completed write. And since at most f faulty servers can send an old timestamp to mislead the reader, choosing the $(2f + 1)$st smallest timestamp rejects all old values. But this condition is not enough because a faulty server can send a higher timestamp too which was never written. The valid() condition deals with this case. Valid($\langle v, ts, wid \rangle$) ensures that a chosen value is not a corrupted one sent by a Byzantine server. This is done by checking if this value was sent by at least $f + 1$ servers. This implies that at least one non-faulty server sent this value and so it is not a corrupt value.

notOld() and fwded() This condition helps the read return a value which is currently being written. The Readers set maintained by each server is used in this part. When a server receives the second phase WRITE_TS message from a writer, it sends a FWD and TS message to all ongoing reads. If a reader receives $f + 1$ messages with the same value $\langle v, ts, wid \rangle$, it can be concluded that it is a valid value currently being written. Again waiting for $f + 1$ same responses implies that at least one correct server forwarded this value. But only checking

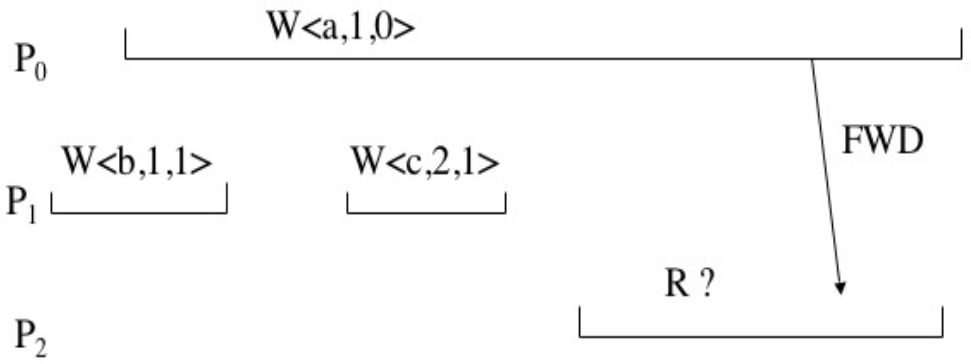

Fig. 4. An execution with a slow writer forwarding messages

for validity is not enough because it is possible that a slow writer is now sending FWD messages for value which has an older timestamp. An example of such an execution is in Fig. 4. To avoid such situations, we add the notOld() check for timestamp.

2. Writer

When a writer starts writing a new value, it has to first decide the timestamp value of the new write. In case of a single writer, this is not an issue as the writer can keep a counter of the number of writes done and use it as a timestamp. For the multi-writer case, one option is to do a read to get the last written timestamp. We use that scheme in our protocol. Since more than one writer could be writing at the same time, the timestamp could be the same and so we include the writer-id to form a pair $\langle ts, wid \rangle$.

The writer's protocol is described in Algorithm 3. In phase 1, the writer sends a WRITE_VAL message to all servers and waits for $n - f$ acknowledgments. At the server side, the variable Rval is updated if the received timestamp is greater than its timestamp. The server sends an acknowledgment even if it does not update Rval. In

Read()

∀ s: timeStamps[s]=⊥ , Fwd[s]=⊥, Values[s]= ∅

//*Phase R1:* **Send**(GET_TS) to all servers

On receive Message from server s

Message: (TS, $\langle ts, wid \rangle$)

timeStamps[s]=$\langle ts, wid \rangle$

Message: (FWD, $\langle v, ts, wid \rangle$, Vals)

Fwd[s]=$\langle v, ts, wid \rangle$

Values[s].add(Vals)

if($|\{s : timeStamps[s] \neq \perp\}| \geq n - f$)

Go to Phase R2

//*Phase R2 :* **Send** (GET_VAL) to all servers

On receive Message from server s

Message: (TS, $\langle ts, wid \rangle$)

timeStamps[s]=$\langle ts, wid \rangle$

Message: (VAL, Val)

Values[s].add(Val)

Message: (FWD, $\langle v, ts, wid \rangle$, Vals)

Fwd[s]=$\langle v, ts, wid \rangle$

Values[s].add(Vals)

if(termination_condition is true for any $\langle v, ts, wid \rangle$)

Send REMOVE_READER to all servers

return $\langle v, ts, wid \rangle$

Algorithm 1: Reader's protocol

Termination Condition

fwded($\langle v, ts, wid \rangle$**)** $\equiv |\{s : Fwd[s] = \langle v, ts, wid \rangle\}| \geq f + 1$

valid($\langle v, ts, wid \rangle$**)** $\equiv |\{s : \langle v, ts, wid \rangle \in Values[s]\}| \geq f + 1$

notOld($\langle v, ts, wid \rangle$**)** $\equiv |\{s : timeStamps[s] \leq \langle ts, wid \rangle\}| \geq 2f + 1$

A read returns if either of the two conditions is true for any value

1. valid($\langle v, ts, wid \rangle$) AND notOld($\langle v, ts, wid \rangle$)
2. fwded($\langle v, ts, wid \rangle$) AND notOld($\langle v, ts, wid \rangle$)

Algorithm 2: Reader's termination condition

phase 2, the writer sends WRITE_TS message to all servers. Each server updates the Rts variable and also sends FWD and TS message to ongoing reads.

3. Server

The server's protocol is described in Algorithm 4. Each server stores two variables **Rval** and **Rts** corresponding to the most recently written value and timestamp. Rval holds the value,timestamp, writer_id triplet and Rts represents the timestamp pair $\langle ts, wid \rangle$. In case of no ongoing write, all non-faulty servers will have the same timestamp pair in Rts and Rval. When there are ongoing writes, the time stamp could be different in Rval and Rts. The key reason for storing value and timestamp separately is that the read and write protocols are two phase. A write first writes the value and then the timestamp but a read first reads the timestamp and then the value. This two phase scheme makes sure that if a reader receives a timestamp, the value corresponding to it is already written at the servers.

Each server also maintains a list of ongoing readers in the **Readers** set. It

Write(v)

//A complete read is done to decide the timestamp

$\langle w, ts, wid \rangle$=Read()

$ts = ts + 1$

//Phase W1

Send(WRITE_VAL,$\langle v, ts, ID \rangle$) to all servers

wait for $(n - f)$ WRITE_ACK1.

//Phase W2

Send (WRITE_TS,$\langle v, ts, ID \rangle$) to all servers

wait for $(n - f)$ WRITE_ACK2

Return ACK

Algorithm 3: Writer's protocol

uses this set to help readers terminate in case of concurrent writes. When a write reaches second phase and sends WRITE_TS message to the server, the server updates it Rts variable and also sends FWD and TS message to all active readers. A new reader is inserted into the Readers set when the server receives a GET_TS message. A reader is removed from the Readers set when the read terminates and sends a REMOVE_READER message to the server.

The changes we made in the algorithm in [7] were :

Last Three Values: The protocol in [7] stored the last three written values at each server which were Rval, Rprev and Rprev2. This was required in their case as they were implementing atomicity. In our protocol, only the most recent value (Rval) is stored at each server.

GetConcurrentReaders: In [7], Aiyer et al. design a GetConcurrentReaders() protocol used by a writer to get the reads concurrent with it. A write executes this protocol in first phase to get the set CR of reads concurrent with it. In the second phase of write protocol, the writer sends the set CR to each server asking it to forward messages to readers in CR.

In our protocol, a writer just signals every server to forward messages to all readers in the Readers set maintained by each server. The difference here is that each correct server might not necessarily forward messages to the same set of readers. But the advantage is we save on time and message complexity. We are yet to figure out if the GetConcurrentReaders() protocol was required in their case.

Forwarded Value: When a server receives Write_TS message from a writer w, it forwards the value being written by the writer w to the ongoing readers. In the

Server()

Readers=$\emptyset$, Rts=$\langle 0,0,0\rangle$, Rval=$\langle 0,0,0\rangle$

//Write Protocol Messages

On receive(WRITE_VAL,$\langle v, ts, wid\rangle$) from writer w

if ($Rval.ts < ts$)

Rval=$\langle v, ts, wid\rangle$

Send WRITE_ACK1 to writer w

On receive(WRITE_TS, $\langle v, ts, wid\rangle$) from writer w

if ($Rts.ts < ts$)

Rts=$\langle ts, wid\rangle$

for each r in Readers

Send (FWD, s, $\langle v, ts, wid\rangle$, $\{\langle v, ts, wid\rangle$, Rval$\}$) to r

Send (TS, s, Rts) to r

Send WRITE_ACK2 to writer w

//Read Protocol Messages

On receive (GET_TS) from reader r

Readers.insert(r)

Send (TS, s, Rts) to r

On receive(GET_VAL) from reader r

Send(VAL, s, Rval) to r

On receive(REMOVE_READER) from reader r

Readers.erase(r)

Algorithm 4: Server's protocol

algorithm in [7], the server sends the latest value stored at the server instead of the value being written by the writer. This was the key change in the algorithm as it helped us develop the multi-writer solution without any bounds on the number of concurrent writers. In case of concurrent writes Rval could be different at the correct servers. While this change helps in terminating a read in case of concurrent writes, it compromises on atomicity.

Forwarding Timestamp: We make one more change in the forwarded part of the algorithm. The server also sends a TS message including the latest timestamp to ongoing reads. This is in addition to the FWD message it sends. This change was required as the reader had a notOld() check on forwarded values too.

E. Analysis

This section discusses the results of simulation and theoretical analysis of the algorithm.

1. Simulation

In order to test the correctness and performance of the algorithm we have simulated it in C++. A simulator class controls the communication between the servers and clients. The protocol is implemented in an event-driven way. Specifically, a new event is created when a message is to be sent and each event is assigned an occurrence time. The events are executed by the simulator in order of their occurrence times. To simulate an asynchronous behavior, random delay was added in the occurrence time of each event. The message is received by the receiver when the corresponding event is executed. The Byzantine behavior of the servers is modeled by randomly picking up f servers which send random values in response to request for timestamp

and value. The input parameters to the code are number of servers, readers, writers and the maximum number of server failures.

The protocol was tested for different schedules and varying network sizes to test termination and correctness. The simulation helped us fix small errors in the algorithm and test it robustly for varying set of parameters. All schedules generated by the simulation satisfied MWReg.

2. Proof of Correctness

a. Wait-Free

The protocol is wait-free in the sense that every client operation completes independent of the behavior of other clients. The following theorems prove that the read/write operation terminate.

Lemma 1. *If no write is concurrent with a read operation, the read operation terminates.*

Proof. At all non-faulty servers, Rval and Rts are updated when a writer writes a value with timestamp greater than Rts. A writer updates Rval in Phase 1 and then Rts in Phase 2. So if there is an ongoing write operation which has completed Phase 1 but not Phase 2, then Rval and Rts could hold different values. But if there is no ongoing write operation, Rval and Rts hold the value of the last completed write with highest timestamp . So when the reader r receives responses from all correct servers for GET_TS and GET_VAL messages, valid() and notOld() will be true for Rval. Thus the reader will terminate and return Rval. □

Theorem 1. *A read operation always terminates.*

Proof. A read operation r will not terminate if neither of these conditions is ever true for any variable val $=\langle v, ts, wid\rangle$

1. notOld(val) AND valid(val)
2. notOld(val) AND fwded(val)

It follows from Lemma 1 that a read operation terminates if no write is concurrent with it.

In case of concurrent writes, let W be the set of writes concurrent with this read and V be the set of values being written by these writers. Since the read has not terminated yet, the writes will ask the servers to send FWD and TS messages to the read r. The FWD messages include the ongoing write's value and Rval, while the TS message includes Rts. Since all correct servers will send the FWD and TS messages, the reader will receive at least $f + 1$ FWD message for each value $v_i \in V$. So fwded(val) will be true for all these values. The only way read r cannot terminate is when notOld(val) is false for all values forwarded by ongoing writes.

This is possible only when the timestamp value(Rts) sent by correct servers is not equal to any of TS(w_i) for $w_i \in W$. This will happen only when Rts $>$ TS(w_i) for $\forall w_i \in W$ at all correct servers. Since Rts was not updated with timestamps of ongoing writes, Rval was not updated either. So the reader will terminate because it will get enough messages for Rval to make it valid and notOld. □

Theorem 2. *A write operation always terminates.*

Proof. A write operation first does a read to decide timestamp. We have already proved that a read operation always terminates. So here we will prove termination of the two phases of the write protocol. In each phase the writer waits for $n - f$ responses from servers. When a non-faulty server receives a message from a writer it sends an acknowledgment even if it does not update its variables. Similarly for phase W2, the server sends FWD messages to readers but does not wait for any response and sends an acknowledgment to the writer. Since at most f servers can be faulty

and may not respond, it is evident that a write operation receives $n - f$ responses in each phase and terminates. □

b. Correctness

In this section, we prove that all executions generated by the algorithm satisfy MWReg.

Theorem 3. *For any read operation r and any write operation w: if $w <_\sigma r$, then TS(w) $\leq$ TS(r).*

Proof. After a write operation w has completed, $n - f$ servers will have Rts $\geq$ TS(w). This implies that at most f correct servers can have timestamp TS$<$ TS(w). Also f Byzantine servers can send old value. So any TS $<$ TS(w) will be received from at most $2f$ servers. So notOld(TS) will never be true for any TS $<$ TS(w).

A read operation r returns a value with timestamp TS only when notOld(TS) is true and so r's timestamp is no less than w's. □

Theorem 4. *For any two write operations w_1 and w_2 : if $w_1 <_\sigma w_2$, then TS(w_1) $<$ TS(w_2).*

Proof. Since a writer does a complete read to decide timestamp, from Theorem 3, TS(r) $\geq$ TS(w_1). The writer increments this timestamp to write the new value and so it follows that TS(w_2) $>$ TS(w_1). □

Theorem 5. *The write operations performed using Algorithm 3 are totally ordered by timestamp and this total order extends $<_\sigma$.*

Proof. Every writer stores the last timestamp value it wrote and also does a read to get the latest timestamp value. It then chooses the maximum of these two values and increments it to assign the timestamp to the new write. So the timestamp of

every writer is unique because the writer appends its ID also to the timestamp. The ordering of these timestamps is defined in 2.1. From Theorem 4, it follows that the ordering of timestamps is consistent with $<_\sigma$. □

Theorem 6. *The algorithm implements MWReg.*

Proof. It has been proved in Theorem [3] that an algorithm that has the properties proved in Theorems 3, 4, 5 implements MWReg. □

In Fig. 5, we show how the algorithm generates the schedule shown in Fig. 1. Since the schedule in Fig. 1 satisfies MWReg, but not atomicity, it follows that our algorithm is not as strong as atomicity.

c. Boundedness

The following two theorems prove that the protocol has bounded complexity. We measure time complexity in terms of rounds where one round represents the maximum time delay in one round trip of communication between a source and a destination. Table II summarizes the time and message complexity.

Theorem 7. *The read protocol has bounded complexity.*

Proof. Theorem 1 proves that the two phases of the read protocol terminate. It follows that the time complexity is 2 rounds. The number of messages generated in each phase is $2n$ and so the total message complexity is $4n$. Since only constant variables of type $\langle v, ts, wid \rangle$ are included in any message, the size of messages is constant. □

Theorem 8. *Any write operation has bounded message and time complexity.*

Proof. It is proved in 2 that the two phases of the write protocol terminate. Since the time complexity of each phase is 1 round, it follows that any write operation takes 2

Po	P1	P2	P3
Start W(x,1) Read=<0,0,0> IncTs(0); ts=1 update({A,B,C},Rval,<1,1,0>) update({A,B,C},Rts,<1,0>) **ack**	**Start W(x,2)** Read=<0,0,0> IncTs(0); ts=1 update({A,C,D},Rval,<2,1,1>) update({A,C,D},Rts,<1,1>) **ack**		
		Start R_1(x) TimeStamps={<1,1>,<1,1>,<1,1>} notOld<2,1,1>&Valid<2,1,1> from {A,B,C} **return 2**	**Start R_2(x)** TimeStamps={<1,1>,<1,1>,<1,1>} notOld<2,1,1>&Valid<2,1,1> from {B,C,D} **return 2**
		Start R_3(x) TimeStamps={<1,1>,<1,1>,<1,0>} notOld<2,1,1>&Valid<2,1,1> from {A,B,C} **return 2**	
	Start R_4(x) TimeStamps={<2,3>,<2,3>,<1,1>} from {A,B,C} Fwd[A]=<4,2,3>,Fwd[B]=<4,2,3> **Return 4**		**Start W(x,4)** notOld<2,1,1>&Valid<2,1,1> IncTs(1); ts=2 update({A,B,C},Rval,<4,2,3>) **Phase 2** update({A,B},Rts,<2,3>) {A,B} send FWD,TS to R_4
	Start R_5(x) TimeStamps={<1,1>,<1,1>,<1,1>} from {B,C,D} where B is corrupted. Values[B,D]=<2,1,1> notOld<2,1,1>&Valid<2,1,1> **return 2**	**Start W(x,3)** notOld<2,1,1>&Valid<2,1,1> -same way as for R_5 IncTs(1); ts=<2,2> !update({A,B,C},Rval,<3,2,2>) **Phase 2** !update({A,B,C},Rts,<2,2) **ack**	Update(C, Rts, <2,3>) **ack**
Start R_6(x) TimeStamps={<2,3>,<2,3><2,2>} Values[A,C]=<4,2,3> notOld<4,2,3>&Valid<4,2,3,> **return 4**			

Fig. 5. An execution of the algorithm that generates the schedule in Fig. 1

rounds. The size of any message generated during the execution of the write operation is constant because only variables of type $\langle v, ts, wid \rangle$ are passed as a parameter in any message. The number of messages generated by a write operation is $4n + 2|R|$ where $|R|$ is the number of read operations concurrent with the write. Since a writer first does a read, the total complexity of a write operation includes that of a read operation too. □

Table II. Complexity of the read/write operations

–	$read()$	$write()$
Rounds	2	4
Messages	$4n$	$8n + 2\|R\|$
Size of Message	Constant	Constant

The storage cost at the servers is constant in our protocol as each server stores only the last written value. Most of the multi-writer register simulations store the last written value by each writer at all servers, and thus have storage cost proportional to the number of writers.

CHAPTER III

CAPACITY RESERVATION

Establishing a connection of a given bandwidth between a source s and a destination d in a communication network is an important network design problem. The related literature gives many approaches to solve this problem. One of the possibilities is to use flow-based algorithms [12] for capacity reservation. However, such algorithms are not designed with a reliability in mind, so the resulting topology might include one or more edges whose failure disconnects the network. In particular, this solution may reserve capacities on only one path between the source and destination. So if any edge on this path fails, the communication link between the source and the destination is broken. Thus, capacities reserved by a min-cost flow algorithm are not resilient to edge failures.

We address this problem in our work and present a resilient capacity reservation algorithm. Our solution is tolerant to crash failure of edges. We assume that at most one edge can fail at any time which is a common assumption in practice. The motivation for resilient capacity reservation is to minimize the loss of network traffic in the event of an edge failure. Our capacity reservation scheme can be combined with network coding [13] to provide instantaneous recovery from edge failures. Fig. 6 shows an example of applications of network coding for instantaneous recovery from edge failures.

Our goal is to minimize the total cost of reserved capacity at all links in the network under the assumption that the cost of each edge is proportional to the capacity used on it. The total cost of a capacity reservation algorithm is the sum of the cost of capacities reserved on all edges in the network. An optimal solution in such a setting reserves capacities on edges in a way that the total cost is minimized, while ensuring

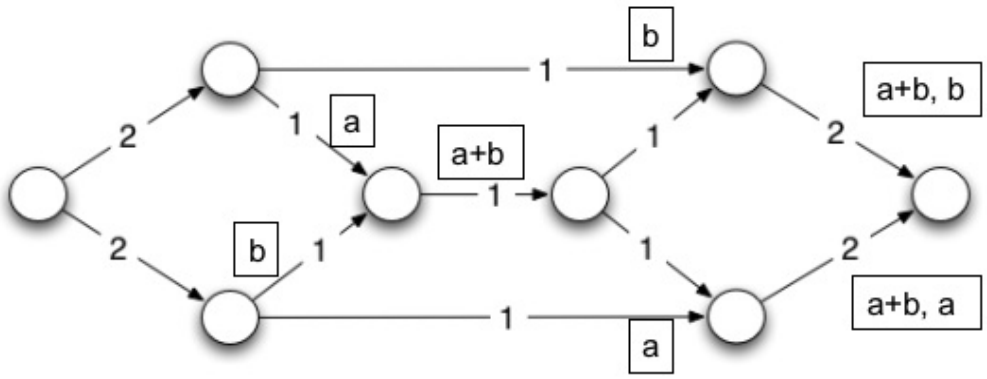

Fig. 6. A network which shows how network coding provides instantaneous recovery from edge failures

that the traffic demand between s and d are met even when an edge fails.

If there are no constraints on the capacities that can be reserved on any edge, there are combinatorial algorithms [14] which give the optimal solution. However, real world networks have upper bounds on capacities and require the capacities to be reserved in integral amounts only. In [14], Brightwell et al. prove that the capacity reservation problem with the integrality constraint is NP-complete. This result also holds for the case with no upper bounds on edge capacities.

We focus on the case of provisioning two units of flow from s to d subject to both the constraints i.e. upper bounds and integrality. It is not known whether this case of the problem is NP-compelete, but to the best of our knowledge, we give the first algorithm to solve this problem and prove that the total cost of our solution is at most four times the optimum. Our solution uses the min-cost flow algorithm from [12] as a building block. We observe that if the upper bound on each edge is one unit, then the optimal solution is to find a flow of value three between the source and destination as it will give three disjoint paths. So if an edge on any path fails in this case, the other two paths are not affected. But when the upper bound on edge capacity is two units

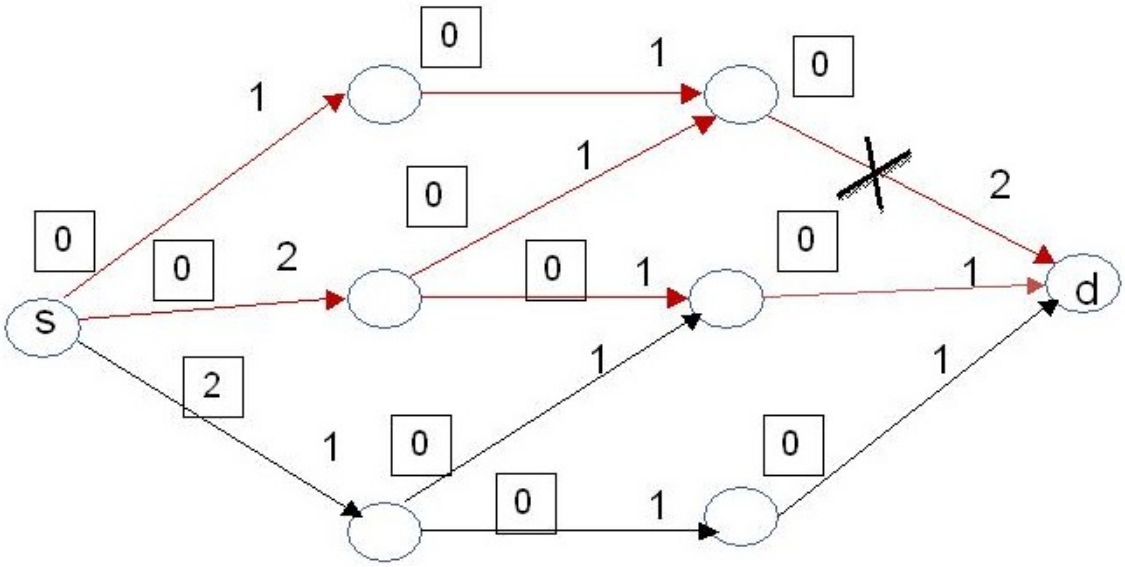

Fig. 7. A network with upper limit on edge capacities and costs (in squares)

or more, then a min-cost flow of value three might not include three disjoint paths as in the example network in Fig. 7. This example shows that finding a flow of three units will not always allow to send two units of flow with instantaneous recovery from a single edge failure.

A. Related Work

There are many approaches to the general problem of capacity reservation in networks. Some of them are theoretically optimal but do not ensure polynomially bounded running time like in [15, 16]. Chekuri at al. [17] approach the resilience issue in a slightly different way by giving a primary and backup path. In a recent work, Grosan et al. [18] extend this approach while optimizing multiple objectives. In [14], Brightwell et al. give combinatorial algorithms for reserving edge capacities in case of no upper bounds on edge capacities. They also discuss many sub-problems for the

case of upper bounds on edge capacities in the paper [19].

B. Model

We model the communication network by a directed graph $G = (V, E)$ with a given source $s \in V$ and destination $d \in V$. Each edge e_i has upper bound on capacity denoted as λ_i. The cost of using one unit of capacity on edge e_i is c_i. The capacity reserved on any edge is termed r_i. The capacity reservation problem can be formulated as the following integer problem :

$$\text{Min (Total Cost} = \sum_{e_i \in E} c_i r_i) \tag{3.1}$$

subject to:

$$r_i \leq \lambda_i \tag{3.2}$$

$$r_i \in Z_i \tag{3.3}$$

where Z_i is the set of integers and for every cut C in the network,

$$\sum_{e_i \in C} r_i \geq 2 + \max(r_i) \tag{3.4}$$

C. Algorithm

The capacities on network edges are reserved in four steps which are summarized in Algorithm 5. The idea is to treat an edge of two units as two separate arcs, that can transmit one unit each. Failure of an edge implies failure of the corresponding arcs. The first step is to find a min-cost flow of value three. This flow is then decomposed into three arc-disjoint paths P_1, P_2, P_3. An example network in Fig. 8 shows an example of such a decomposition.

A flow of three units is required because we loose at least flow of one unit when an

Step 0: Substitute each edge of capacity two by two parallel arcs, and each edge of capacity one by one arc

Step 1: Find a min-cost flow of value three from s to d. Decompose it into three arc-disjoint paths P_1, P_2, P_3

Step 2: Goal: Protect edges in P_1, P_2

Reverse arcs in P_1, P_3 and arcs in P_2 which share an edge with P_1. Set the cost of reversed arcs to zero. Let the resulting graph be G'. Find a flow of value one in G'.

Step 3: Goal: Protect edges in P_2, P_3

Reverse arcs in P_2, P_1 and arcs in P_3 which share an edge with P_2 . Set the cost of reversed arcs to zero. Let the resulting graph be G'. Find a flow of value one in G'.

Step 4: Goal: Protect edges in P_3, P_1

Reverse arcs in P_3, P_2 and arcs in P_1 which share an edge with P_3. Set the cost of reversed arcs to zero. Let the resulting graph be G'. Find a flow of value one in G'.

Algorithm 5: Capacity reservation algorithm to provide resilient flow of two units

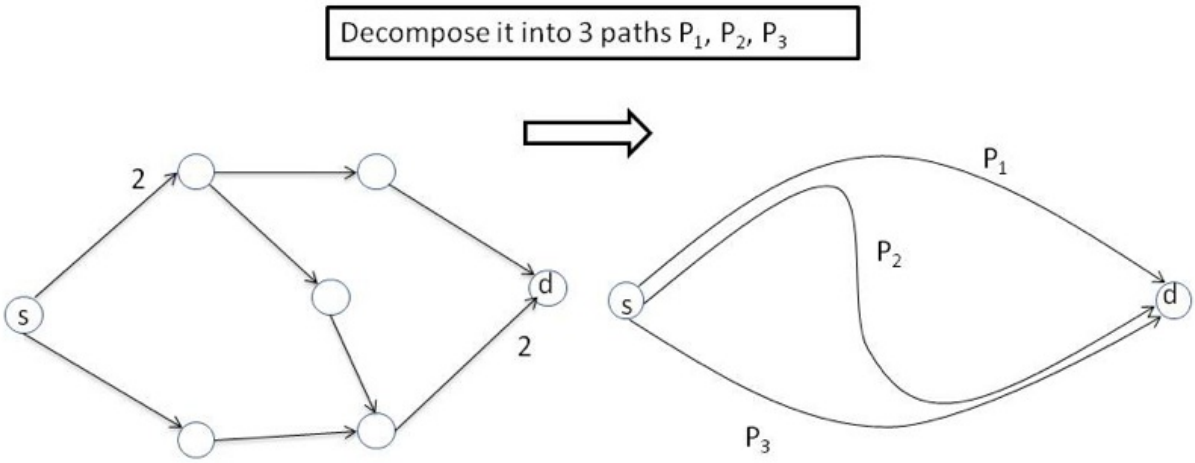

Fig. 8. An example network to show how flow of value three is decomposed into three paths

edge fails. If the capacity reserved on any edge e_i is 2, the failure of that edge results in loss of two units of flow. So we need to reserve additional capacity to tolerate the failure of edge e_i. We refer to this as "protecting" edge e_i.

Steps 2, 3 and 4 protect all edges of capacity 2. Such an edge is included in any 2 of the three paths P_1, P_2, P_3. Step 2 deals with edges that appear in both P_1 and P_2, Step 3 deals with edges that appear in both P_2 and P_3, and Step 4 deals with edges that appear in P_1 and P_3. The idea behind protecting an edge is to reverse a set of edges and then find a flow of 1 unit in the residual graph. Fig. 9 shows which set of edges are reversed to protect edges common in P_1 and P_2. An example with 1 unit of flow in the residual graph is depicted in Fig. 10. The solution graph includes the edges added in each step of the algorithm.

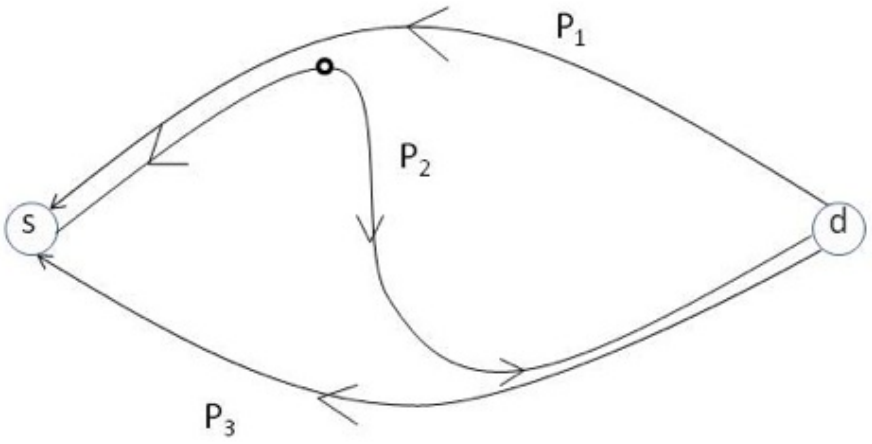

Fig. 9. An example graph G'

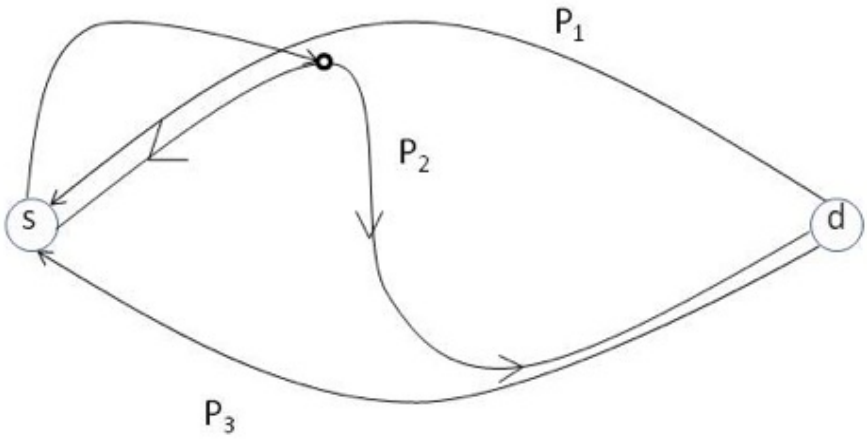

Fig. 10. An example network to demonstrate step 2 of algorithm 5

D. Proof of Correctness

We show that the total cost of the capacities reserved by our algorithm is at most four times that of the optimal solution. Let the total cost of the capacities reserved be OPT in the optimal solution. $C(P)$ denotes the total cost of edges in path P and G^{OPT} represents the optimal graph. The optimal graph is formed from the edges that were assigned positive capacities in the optimal solution.

The main idea of the proof is to show that the cost of the edges we add in each step is at most OPT. The capacities are reserved in a way that a flow of two units is guaranteed between the source s and the destination d even when an edge fails. In the worst case, one edge failure could result in loss of 1 unit of flow from s and d. So the optimal graph will at least have a flow of value three from s and d. The step 1 of our algorithm is to find a min-cost flow of value three. So the cost of capacity reservation performed in Step 1 is at most OPT. In what follows, we give an analysis of Step 2 of the Algorithm and show that the cost of additional capacity in Step 2 is at most OPT. The same reasoning can be applied for Steps 3 and 4.

Lemma 2. *In the graph G' in Step 2 of the algorithm, there exists a path P with $C(P) \leq OPT$.*

Proof. Consider a sub-graph $G'' \subseteq G'$ defined as follows :
$G'' = (E(P_1 \cup P_2 \cup P_3) \cup G^{OPT}) \cap G' \cup$ {all edges in G' of zero cost } where G^{OPT} is the optimal graph. The graph G'' includes all zero cost edges from G' and the edges which are common in G' and G^{OPT}. It follows that the cost of the edges in G'' will be less than OPT because it only includes zero cost edges and an intersection of edges in G^{OPT}.

We need to prove that there is a path from source s to destination d in the graph G''. This part is proved in the next lemma. □

Lemma 3. *G'' has a path from source s to destination d.*

Proof. We prove that in the graph G'', every (s, d)-cut has a flow of value one in the forward direction which is also part of the optimal graph. We consider the three following cases:

Case 1: C does not include any edge which contains an arc from P_1 and P_2 or P_2 and P_3.

This case is trivial because edges of P_2 are in the forward direction with flow of value one. Also these edges belong to the optimal solution and so we are not paying more than OPT.

Case 2: C includes only one edge e that contains an arc from P_1 and an arc from P_2.

This case has three possible sub-cases related to the intersection of P_3 :

Case 2a : C intersects P_3 only once as shown in Fig. 11. In this case, the optimal graph G^{OPT} will also have an edge in the forward direction with capacity 1 to make it robust to failure of edge $P_1 \cap P_2$ of capacity 2.

Case 2b: C intersects P_3 more than once as in Fig. 12. In this case, since flow P_3 is reversed, it must intersect the cut in the forward direction once to ensure a continuous flow.

Case 2c: C includes at least one edge that contains an arc from P_2 and an arc from P_3 as in Fig. 13

In this case, since we did not reverse the edges of P_2 which were intersecting with P_3, we have a flow of value one in the forward direction in P_2 which belongs to optimal graph.

Case 3: C includes more than one edge that contains an arc from P_1 and P_2 as in

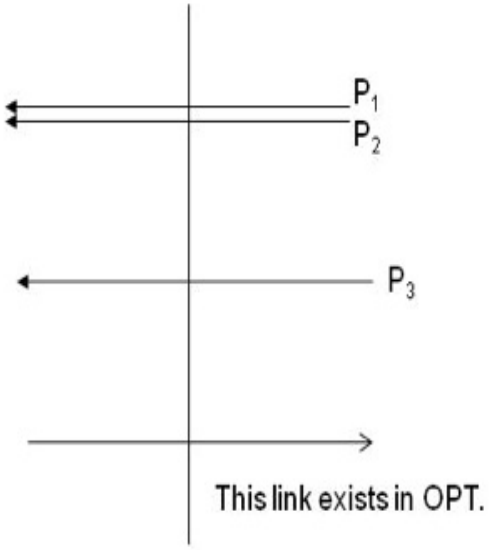

Fig. 11. Cut of Case 2a

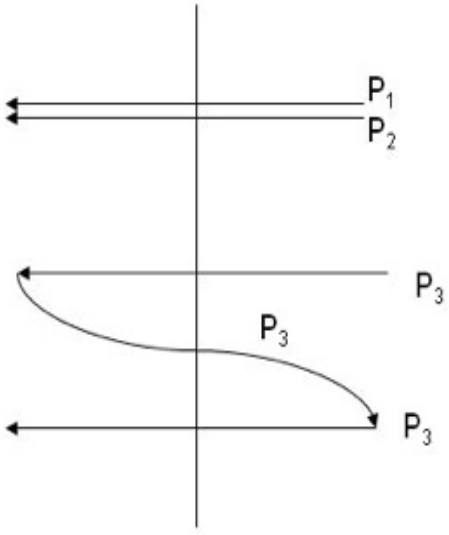

Fig. 12. Cut of Case 2b

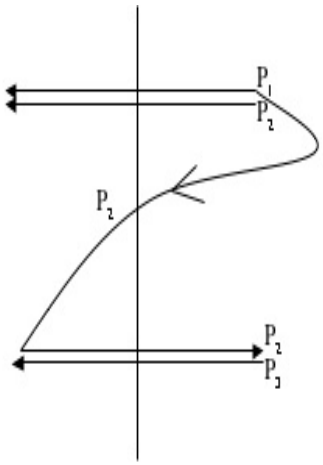

Fig. 13. Cut of Case 2c

Fig. 14.

In this case, since flow P_1 is reversed, it must intersect the cut in the forward direction once to ensure a continuous flow.

□

Theorem 9. *Algorithm 5 is an approximate solution with cost at most four times optimal.*

Proof. It follows from lemmas 2 and 3 that the edges we add in Step 2 of the algorithm have cost at most OPT. The same logic applies to Steps 3 and 4. So the total cost of the solution is $4 \cdot \text{OPT}$. □

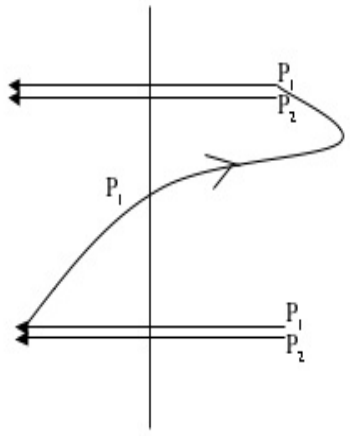

Fig. 14. Cut of Case 3

CHAPTER IV

CONCLUSION

With the increasing importance of secure communication, research in the direction of fault-tolerant algorithms has gained considerable attention. In this thesis, we give fault-tolerant algorithms for two models.

The first one is a distributed storage protocol, which implements a multi-writer multi-reader register over a set of servers, a fraction of which can be Byzantine. Our solution is optimally resilient (i.e. it works for $n \geq 3f+1$ which has been proved to be minimal), has bounded message and time complexity and does not put a constraint on the number of writers. The client protocols for both readers and writers are wait-free and no communication is required between the servers. Theoretical analysis and simulation were done to prove the correctness of the protocol and test its performance.

The second model is a network with a given source and a destination and costs associated with per unit bandwidth usage on any edge. Each edge also has a maximum capacity limit. We give a capacity reservation algorithm which is tolerant to crash failure of one edge and achieves approximation ratio four.

A. Future Work

A replication-based multi-writer implementation which satisfies atomicity and does not put a bound on the number of writers is still an open problem. It would be interesting to investigate a hybrid scheme combining replication and erasure coding. Along the direction of results in [3], adding the WriteBack block to the protocol might result in an algorithm satisfying a stronger consistency condition. This could give some insight for proving a separation with respect to complexity between multi-writer atomicity and multi-writer regularity. Also using Byzantine quorums to plug into the

algorithms from [3] to give fault-tolerant versions of all the consistency conditions is an interesting direction.

In the area of resilient capacity reservation, proving that the problem we address is NP-complete or designing a polynomial time algorithm which gives optimal solution is an open issue. Another issue is to design algorithms tolerant to Byzantine failure of edges. Extending the protocol to the case of providing 3 units or more of reliable communication link between a source and a destination is an open problem. Another interesting direction is to extend the algorithm to the multi-cast case where there are multiple destinations.

REFERENCES

[1] F. Eisenbrand, F. Grandoni, G. Oriolo, and M. Skutella, "New approaches for virtual private network design," in *Automata, Languages and Programming*, Philadelphia, PA, USA, 2005, pp. 1151–1162.

[2] L. Lamport, "On interprocess commnunication, Part I: Basic formalism," *Distributed Computing*, vol. 1, pp. 77–85, 1986.

[3] C. Shao, E. Pierce, and J. L. Welch, "Multi-writer consistency conditions for shared memory objects," *Distributed Computing*, vol. 2848, pp. 106–120, 2003.

[4] J. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine storage," in *Proceedings of the International Symposium on Distributed Computing*, London, UK, 2002, pp. 311–325.

[5] R. Guerraoui and M. Vukolic, "How fast can a very robust read be?," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, New York, NY, USA, 2006, pp. 248–257.

[6] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, "Byzantine disk paxos: Optimal resilience with Byzantine shared memory," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, New York, NY, USA, 2004, pp. 226–235.

[7] A. S. Aiyer, L. Alvisi, and R. A. Bazzi, "Bounded wait-free implementation of optimally resilient Byzantine storage without (unproven) cryptographic assumptions," *Distributed Computing*, vol. 4731, pp. 7–19, 2007.

[8] R. Bazzi and Y. Ding, "Non-skipping timestamps for Byzantine data storage systems," in *Proceedings of the International Symposium on Distributed Computing*, London, UK, 2004, pp. 405–419.

[9] R. Bazzi and Y. Ding, "Bounded wait-free *f-resilient* atomic Byzantine data storage systems for an unbounded number of clients," in *Proceedings of the International Symposium on Distributed Computing*, London, UK, 2006, pp. 299–313.

[10] C. Cachin and S. Tessaro, "Optimal resilience for erasure-coded Byzantine distributed storage," in *International Conference on Dependable Systems and Networks*, Washington, DC, USA, 2005, pp. 115–124.

[11] R. Guerraoui and M. Vukolic, "Refined quorum systems," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, New York, NY, USA, 2007, pp. 119–128.

[12] A.K. Ahuja, T.L. Magnanti, and J.B.Orlin, *Network Flows- Theory, Algorithms, and Applications*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[13] R. Koetter and M. Médard, "An algebraic approach to network coding," *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 782–795, 2003.

[14] G. Brightwell, G. Oriolo, and F. B. Shepherd, "Reserving resilient capacity in a network," *SIAM Journal on Discrete Mathematics*, vol. 14, pp. 524–539, 2001.

[15] D. Bienstock and G. Muratore, "Strong inequalities for capacitated survivable network design problems," *Mathematical Programming*, vol. 89, pp. 122–147, 2001.

[16] A. Balakrishnan, T. Magnanti, J. Sokol, and Y. Wang, "Spare-capacity assignment for line restoration using a single-facility type," *Operations Research*, vol. 50, no. 4, pp. 617–635, 2002.

[17] C. Chekuri, A. Gupta, A. Kumar, J. Naor, and D. Raz, "Building edge-failure resilient networks," in *Proceedings of the 9th Integer Programming and Combinatorial Optimization Conference*, London, UK, 2002, pp. 439–456.

[18] C. Grosan, A. Abraham, and B. Helvik, "Building multiobjective resilient networks," in *International Conference on Computer Modelling and Simulation*, Washington, DC, USA, 2008, pp. 204–209.

[19] G. Brightwell, G. Oriolo, and F. B. Shepherd, "Reserving resilient capacity in a network with upper bound constraints," *Networks*, vol. 41, no. 2, pp. 87–96, 2003.

VITA

Khushboo Kanjani received her B.Tech. degree in electrical engineering from Indian Institute of Technology, Roorkee, India in 2006. She started research work under the guidance of Dr. Jennifer Welch in September 2006. She obtained her M.S. degree in computer engineering in December 2008 from Texas A&M University. Her research interests are distributed algorithms, specifically fault-tolerant distributed storage protocols. She can be contacted at khushboo.kanjani@gmail.com and her mailing address is Khushboo Kanjani, Department of Computer Science, Texas A&M University, College Station, TX, 77843-3112, USA.